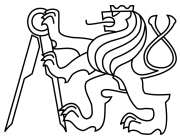
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF BACHELOR'S THESIS

Title:	Automata library - LR parser construction
Student:	Martin Kočička
Supervisor:	Ing. Jan Trávníček
Study Programme:	Informatics
Study Branch:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2016/17

Instructions

Get familiar with the current implementation of the Automata library toolkit[1]. Study LR(0) and SLR(1) parser construction.
Propose suitable extensions of data structures in Automata library toolkit for representing LR(0) and SLR(1) parser.
Implement these extensions of data structures. Extend the library to include LR(0) and SLR(1) parser construction algorithms using these new data structures.
Implement an algorithm of syntactic analysis using the LR(0) and SLR(1) parser.
Test the implemented extensions and algorithms.

References

[1] Martin Žák: Automatová knihovna – vnitřní a komunikační formát. Bakalářská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, Praha, 2014.

L.S.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague January 9, 2016

Czech Technical University in Prague
Faculty of Information Technology
Department of Theoretical Computer Science

Bachelor's thesis

Automata library – LR parser construction

Martin Kočička

Supervisor: Ing. Jan Trávníček

17th May 2016

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on 17th May 2016 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Kočička, Martin. *Automata library – LR parser construction.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2016.

Abstract

This thesis covers basic LR parsing algorithms. We describe bottom-up and shift-reduce parsing methods in general, and then we focus on LR parsing. We go into more detail with two algorithms—LR(0) and SLR(1). Suitable algorithms and data structures are presented and implemented into the Automata library. We also explore existing solutions and the Automata library itself.

Keywords parsing, bottom-up parsing, shift-reduce parsing, LR parsing, SLR parser

Abstrakt

Předmětem této práce jsou základní LR parsovací algoritmy. Práce v úvodu popisuje obecně bottom-up a shift-reduce parsování. Dále se zaměřuje na LR parsování, specificky LR(0) a SLR(1) parsery. Práce obsahuje návrh potřebných datových struktur a algoritmů pro implementaci těchto parserů. Jsou prozkoumána existující řešení, a přidán základní popis Automatové knihovny. Práce je součástí projektu Automatová knihovna.

Klíčová slova parsování, bottom-up parsování, shift-reduce parsování, LR parsování, SLR parser

Contents

List of Figures

List of Tables

Introduction

Parsing (syntax analysis) is one of the most crucial tasks solved in computing. It has wide range of usages—from processing programming languages, structured text such as XML, structured binary data, to DNA pattern recognition and many more.

Parsing algorithms can be split into two categories: top-down and bottom-up. Names of these two methods are referring to parse trees, but it is easier to explain the parsing process using formal grammars. In top-down method, we start with the initial symbol of a grammar, and apply production rules until we produce the input string. On the other hand, in bottom-up method we start with the input string, and replace right-hand sides of the production rules by the nonterminals inside the string, until we reduce it to the start symbol.

This thesis focuses on LR parsing. LR parsers are deterministic bottom-up parsers, which produce a correct parse in linear time, due to the fact that they don't do any guessing or backtracking. LR parsers read input from left to right, construct a rightmost derivation in reverse, and they can detect syntax errors as soon as possible.

The objective of this thesis is to describe selected algorithms, and to implement them into the Automata library. Two algorithms, LR(0) and SLR(1), have been selected, since they are the foundation of almost all other powerful LR parsing methods.

Chapter 1

Theory

1.1 Strings

Definition 1.1.1. *Alphabet* is a finite set of elements (called *symbols*).

Example 1.1.1. Set $\{0, 1\}$ is an alphabet consisting of two symbols.

Definition 1.1.2. *String* is a finite sequence of symbols belonging to the alphabet.

Instead of denoting the strings using the sequence notation, we will just append the symbols to each other.

Example 1.1.2. The string $(\alpha_0, \alpha_1, \alpha_2, \alpha_3)$ will be written as $\alpha_0\alpha_1\alpha_2\alpha_3$.

Example 1.1.3. 0, 01 and 01101 are examples of strings over alphabet $\{0, 1\}$.

Definition 1.1.3. The number of symbols in the sequence is called *length* of the string, and for an arbitrary string w, the length will be denoted $|w|$.

Definition 1.1.4. An empty sequence of symbols is also a string, called *empty string*. It will be denoted using symbol ε.

Definition 1.1.5. Let $x = (x_0, x_1, \ldots, x_{n-1}, x_n)$ and $y = (y_0, y_1, \ldots, y_{m-1}, y_m)$ be strings. String $z = (x_0, x_1, \ldots, x_{n-1}, x_n, y_0, y_1, \ldots, y_{m-1}, y_m)$ is called the *concatenation* of strings x and y, and will be denoted xy.

Example 1.1.4. String 01101 is a concatenation of strings 01 and 101.

Definition 1.1.6. *Language* is a set of strings over an alphabet.

Example 1.1.5. Set $\{0, 01, 01101\}$ is a language over the alphabet $\{0, 1\}$.

1.2 Grammars

Definition 1.2.1. *Grammar* is a quadruple consisting of following elements:

1. Finite set of nonterminal symbols (often denoted N).
2. Finite set of terminal symbols (often denoted Σ).
3. Finite set of production rules of the form $\alpha \to \beta$ where $\alpha \in (N \cup \Sigma)^* N (N \cup \Sigma)^*$ and $\beta \in (N \cup \Sigma)^*$ (often denoted P).
4. *Start symbol* which belongs to N (often denoted S).

It is common to write the rules for the same left-hand side on one line, separating right-hand sides using the pipe symbol "|".

Example 1.2.1. Production rules $A \to \alpha,\ A \to \beta,\ A \to \gamma$ could be written as $A \to \alpha \mid \beta \mid \gamma$.

1.2.1 Context-free grammar

Definition 1.2.2. *Context-free grammar* is a grammar where all the production rules are in format $A \to \alpha$ (A is a single nonterminal symbol and α is a string of nonterminal and/or terminal symbols).

Example 1.2.2 (Expression grammar)**.** An example of context-free grammar is expression grammar, which is used to generate arithmetic expressions with parentheses, addition, and multiplication. Most of the examples in this work use this grammar.

Let $EG = (\{E, T, F\}, \{+, *, (,), id\}, P, E)$.
$P = \{$
$\quad E \to E + T \mid T,$
$\quad T \to T * F \mid F,$
$\quad F \to (E) \mid id,$
$\}$

1.2.2 Derivations

Definition 1.2.3. A *derivation* of a string for a grammar is a sequence of production rule applications that transforms the start symbol into the string. We denote the derivation using the $\Rightarrow$ operator. So $A \Rightarrow B$ means "A derives B". We will be also using $A \overset{*}{\Rightarrow} B$ which means "A derives B in zero or more steps".

Definition 1.2.4. In *leftmost derivation*, we always replace the leftmost nonterminal when applying the rules. We denote leftmost derivation using the operator $\underset{lm}{\Longrightarrow}$ in this work.

Definition 1.2.5. In *rightmost derivation*, we replace the rightmost nonterminal when applying the rules. We denote rightmost derivation using the operator $\underset{rm}{\Longrightarrow}$.

Definition 1.2.6. A *sentential form* is any string that can be derived from the start symbol (even the start symbol itself).

Definition 1.2.7. If $S \overset{*}{\underset{lm}{\Longrightarrow}} \alpha$ (S is a start symbol of grammar G), we say that α is a *left-sentential form* of G.

Definition 1.2.8. If $S \overset{*}{\underset{rm}{\Longrightarrow}} \beta$ (S is a start symbol of grammar G), we say that β is a *right-sentential* form of G.

Definition 1.2.9. A context-free grammar is called *ambiguous*, if it contains multiple leftmost derivations for a single sentential form.

Definition 1.2.10. Let $G = (N, \Sigma, P, S)$ be a context-free grammar and $A \in N$. If there exists derivation $A \overset{*}{\Rightarrow} A\alpha,\ \alpha \in (N \cup T)^*$, nonterminal A is said to be *left recursive*.

Definition 1.2.11. Grammar G is *left recursive* if it contains at least one left recursive nonterminal.

Definition 1.2.12. Production rule from a context-free grammar is *left recursive* if the right-hand side starts with the same nonterminal as the left-hand side. If the production rule is in form $A \to A\alpha,\ \alpha \in (N \cup T)^*$, the production rule is said to be *directly left recursive*.

Definition 1.2.13. Grammar is said to be *indirectly left recursive*, if it is possible, starting from any nonterminal A, to derive in two or more steps a string whose leftmost symbol is A.

1.2.3 Parse tree

Parse tree is used to describe a production rules applied to get a certain sentential form. Unlike the derivation, a parse tree does not state the order of applications.

Definition 1.2.14. *Parse tree* for a context-free grammar G is a rooted tree where the root is labeled by the start symbol of G, internal nodes are labeled by the nonterminals of G, and leaves are labeled by the terminals of G. Children of an internal node E have to be in order of symbols of a right-hand side of some arbitrary production rule with left-hand side E.

The concatenation of leaf nodes in the preorder traversal gives us the sentential form represented by the tree. For an unambiguous grammar, there is exactly one parse tree for each sentential form.

Example 1.2.3. Example of a parse tree for the expression grammar and sentential form $id + id$ is shown in Figure 1.1.

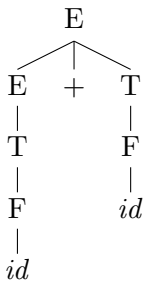


Figure 1.1: Parse tree for the expression grammar and sentential form $id + id$.

1.3 Deterministic finite automaton

Definition 1.3.1. *Deterministic finite automaton* is a quintuple consisting of following elements:

1. Finite set of states (often denoted Q).
2. The alphabet (often denoted Σ).
3. A transition function $Q \times \Sigma \to Q$ (often denoted δ).
4. *Start* (*initial*) state which belongs to Q (often denoted q_0).
5. A set of *final* (*accepting*) states which is a subset of Q (often denoted F).

Example 1.3.1. We usually describe a deterministic finite automaton using a diagram. Let Z be a deterministic finite automaton.

$$Z = (\{a, b, c, d\}, \{0, 1\}, \delta, a, \{d\}).$$

δ is defined by following equations:

$$\delta(a, 0) = b,\ \delta(a, 1) = a$$
$$\delta(b, 0) = b,\ \delta(b, 1) = c$$
$$\delta(c, 0) = d,\ \delta(c, 1) = a$$
$$\delta(d, 0) = d,\ \delta(d, 1) = d$$

Figure 1.2 shows how the diagram for Z would look like.

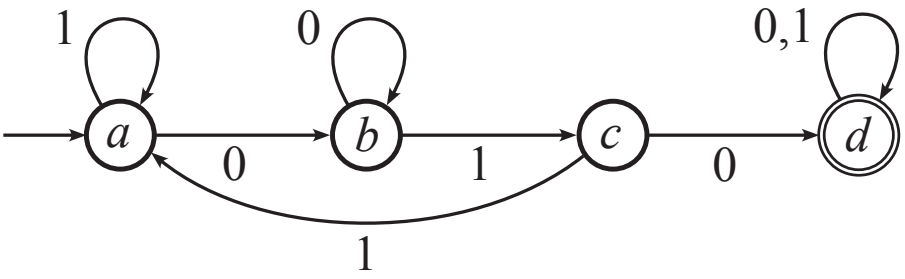


Figure 1.2: Example of a deterministic finite automaton [1].

Chapter 2

Bottom-up parsing

A bottom-up parse methods work by constructing a parse tree from the leaves up, which means we work backward and apply production rules in reverse. In this method we search in the input string until right-hand side of a production rule is recognized, and then the that substring is replaced by the left-hand side of the production rule. As the result, we should reduce the whole string to the start symbol of the grammar. This is the opposite of the top-down parsing, where you derive from the start symbol until you reach the input string. In general, bottom-up parsing algorithms are more powerful than top-down algorithms, which is not surprising since their construction is more complex.

2.1 Reductions

Example 2.1.1. As an example of reductions, we will show a reduction of a string $id + (id * id)$ from the expression grammar to the start symbol. The sequence of these reductions could be following:

$id + (id * id)$, $id + (F * id)$, $id + (T * id)$, $id + (T * F)$, $id + (T)$, $id + (E)$, $id + F$, $id + T$, $F + T$, $T + T$, $E + T$, E

First we reduce the second id to F using the production rule $F \to id$, then F is reduced to T using the production rule $T \to F$, rightmost id is then reduced to F, $T * F$ is then reduced to T using the production rule $T \to T * F$, and so on.

Notice that in some steps we had a choice of multiple reductions, and selecting the right one is the problem we have to solve. Some algorithms like GLR [2] will fork the parsing procedure, and try all possible solutions, but these algorithms will not be discussed in this work.

"By definition, a reduction is the reverse of a step in a derivation (recall that in a derivation, a nonterminal in a sentential form is replaced by the body of one of its productions). The goal of bottom-up parsing is therefore to construct a derivation in reverse." [3, p. 235]

Considering this, the reductions in Example 2.1.1 were representation of following derivation in reverse:
$E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow id + T \Rightarrow id + F \Rightarrow id + (E) \Rightarrow id + (T) \Rightarrow id + (T * F) \Rightarrow id + (T * id) \Rightarrow id + (F * id) \Rightarrow id + (id * id)$

A *handle* is a substring in the input string, which is found on a right-hand side of some rule, so it can be reduced. For this to be a handle, the reduction must be a step in the reverse rightmost derivation.

Definition 2.1.1. Let S be a start symbol of grammar G. If $S \underset{rm}{\overset{*}{\Longrightarrow}} \gamma B w \underset{rm}{\Longrightarrow} \gamma\beta w$, then β is a handle of $\gamma\beta w$ [3, p. 235].

We use handles to construct the rightmost derivation in reverse. Let's say we have input string w, and we want to construct following derivation in reverse.

$$S = \gamma_0 \underset{rm}{\Longrightarrow} \gamma_1 \underset{rm}{\Longrightarrow} \cdots \underset{rm}{\Longrightarrow} \gamma_{n-1} \underset{rm}{\Longrightarrow} \gamma_n = w$$

To construct the rightmost derivation in reverse, we just find the handle of γ_n and reduce it, then we find the handle of γ_{n-1} and reduce it, and so on, until we have reduced the string to the start symbol S.

2.2 Shift-reduce parser

Shift-reduce parser consists of the same two data structures as LL(1) [4] parser: stack and the input string. We either *shift* symbols onto the stack (simultaneously removing them from the input string), or we *reduce* (pop) them from the stack, hence the name *shift-reduce*. We look for the handles on the top of the stack. We will denote both bottom of the stack and the end of the input string using the symbol "$". Sometimes we will refer to the "input string" just as the "input". The parse is successful when the entire input string has been shifted onto the stack and reduced to the start symbol of the grammar [5, p. 25]. The state of the shift-reduce parser after a successful parse of a string from grammar $G = (N, \Sigma, P, S)$ is shown in Table 2.1.

Table 2.1: State of the shift-reduce parser after a successful parse.

Stack	**Input**	**Action**
$\$S$	$\$$	Accept

Example 2.2.1. Let $w = (id + id)$ be a string from the expression grammar. The list of shift-reduce actions with current parser state is shown in Table 2.2.

The sequence of reduce actions forms the rightmost derivation in reverse.

Table 2.2: Shift-reduce parse of a string ($id + id$) from the expression grammar.

Stack	**Input**	**Action**
\$	$(id + id)$\$	Shift
\$(	$id + id)$\$	Shift
\$(id	$+ id)$\$	Reduce $F \to id$
\$(F	$+ id)$\$	Reduce $T \to F$
\$(T	$+ id)$\$	Reduce $E \to T$
\$(E	$+ id)$\$	Shift
\$(E +	$id)$\$	Shift
\$(E + id	)\$	Reduce $F \to id$
\$(E + F	)\$	Reduce $T \to F$
\$(E + T	)\$	Reduce $E \to E + T$
\$(E	)\$	Shift
\$(E)	\$	Reduce $F \to (E)$
\$$F$	\$	Reduce $T \to F$
\$$T$	\$	Reduce $E \to T$
\$$E$	\$	Accept

2.3 Conflicts

When a shift-reduce parser tries to parse using a grammar it cannot handle, the result will be either a *reduce/reduce conflict* or a *shift/reduce conflict*. Reduce/reduce conflict occurs when the parser is unable to decide which production rule use to reduce, and shift/reduce conflict occurs when the compiler is unable to decide whether to shift or reduce. The various shift-reduce algorithms differ in the way in which they handle these conflicts.

Chapter 3

LR parsing

LR is an initialism—**L** states that we read the input from left to right and **R** states that we construct a rightmost derivation in reverse. LR parsers are deterministic bottom-up parsers, and produce a correct parse in linear time. Another useful property of LR parser is that it can detect syntax errors as soon as possible. Names of LR algorithms are usually followed by a number in parenthesis, which states how many symbols ahead are we looking when constructing the parser. The lookahead symbols help us resolve the shift-reduce conflicts.

Unlike LL(1) parsers, LR parsers are really hard to write by hand. There are many parser generators that help us with building a LR parser, and I described a couple of them in Chapter 5, Section 5.1. The goal of this work is to implement a LR parser in the Automata library.

The LR parser consists of input, action and goto tables, stack, and output. The only part where the various LR methods differentiate are the contents of action and goto tables. The stack consists of states, instead of symbols (compared to the general shift-reduce parser). In SLR(1), the stack consists of states of the LR(0) automaton, and it is similar for more powerful algorithms like LALR(1) and LR(1).

Example 3.0.1. An example of LR parser is shown in Figure 3.1. The parser has already seen the input 0110, and there are states (I_1, I_3, I_5, I_1) on the stack (from bottom to top).

3.1 Action table

Action table is indexed by state and a terminal symbol (including the end-of-input character \$), and there are 4 possible types of entries (corresponding to the actions of shift-reduce parser):

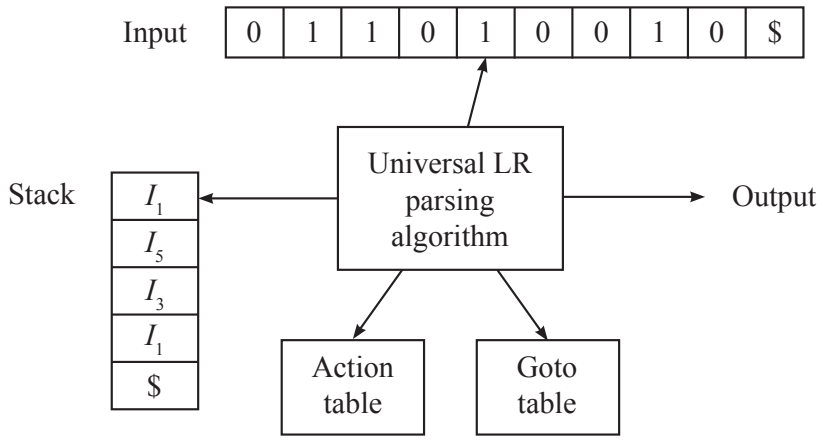


Figure 3.1: LR parser.

- **Shift**[s] Symbol is shifted from the input string, but we actually push state s representing the terminal symbol onto the stack, instead of the symbol itself, as we did with shift-reduce parsers.
- **Reduce**[$A \to \alpha$] We pop $|\alpha|$ states from the stack, and then push $Goto[stack.top(), A]$ (defined in Section 3.2) onto the stack.
- **Accept** The string was parsed successfully.
- **Error** The parser encounters an error and halts. We might want to trigger some kind of recovery procedure. Error action is usually denoted as no value in the corresponding cell.

An entry from the action table will be denoted $Action[state, terminal]$.

Example 3.1.1. The action table for the expression grammar is shown in Table 3.1. For the sake of simplicity, we will be referring to the states using assigned number identifiers. Start state is denoted 0, the other identifiers can be assigned arbitrarily. We will also assign numbers to the rules, so *Reduce[i]* means "reduce by rule i from the following list".

0. $E \to E + T$
1. $E \to T$
2. $T \to T * F$
3. $T \to F$
4. $F \to (E)$
5. $F \to id$

Table 3.1: Action table for the expression grammar.

	+	*	(	)	***id***	**$**
0			Shift[11]		Shift[6]	
1			Shift[11]		Shift[6]	
2			Shift[11]		Shift[6]	
3	Reduce[2]	Reduce[2]		Reduce[2]		Reduce[2]
4	Reduce[1]	Shift[2]		Reduce[1]		Reduce[1]
5	Shift[1]					Accept
6	Reduce[5]	Reduce[5]		Reduce[5]		Reduce[5]
7	Shift[1]			Shift[10]		
8	Reduce[3]	Reduce[3]		Reduce[3]		Reduce[3]
9	Reduce[0]	Shift[2]		Reduce[0]		Reduce[0]
10	Reduce[4]	Reduce[4]		Reduce[4]		Reduce[4]
11			Shift[11]		Shift[6]	

3.2 Goto table

The second table the LR parser uses is the *goto* table. The goto table is indexed by state and a nonterminal symbol, and it is used to decide which state will be pushed onto the stack after a reduction. When we reduce by arbitrary rule $A \to \alpha$, we pop $|\alpha|$ symbols from the stack, and look in goto table where state is the new top of the stack, and nonterminal is A. Then we proceed to push the result onto the stack (or report error). An entry from the goto table will be denoted $Goto[state, nonterminal]$.

Example 3.2.1. The goto table for the expression grammar is shown in Table 3.2. We will be using the same notation for states as we used in Example 3.1.1.

3.3 Parsing algorithm

The LR parsing algorithm is the same for all LR parsing methods (I will be sometimes referring to it as the "universal LR parsing algorithm"), only the contents of action and goto tables differ. This algorithm is described in Algorithm 3.1.

Table 3.2: Goto table for the expression grammar.

	E	***T***	***F***
0	5	4	8
1		9	8
2			3
3			
4			
5			
6			
7			
8			
9			
10			
11	7	4	8

```
Name: parseLR
Input: action table, goto table, input string

push start state onto the stack
let c be the first input symbol

repeat forever
  switch Action[stack.top(), c]
    case Shift[s]
      push s onto the stack
      c is now the next input symbol
    case Reduce[A → α]
      pop |α| items from stack
      push Goto[stack.top(), A] onto the stack
    case Accept
      halt the parse with successful result
    case Error
      halt the parse with error result
```

Algorithm 3.1: Universal LR parsing algorithm

Chapter 4

LR(0) parsing

The first LR method we will look at is LR(0). The zero in parentheses states that we have have no lookahead information while creating the parsing automaton/tables. LR(0) parsers use a deterministic finite automaton for finding the handle (called *LR(0) automaton*). I will show in this chapter how to construct and use this automaton.

4.1 LR(0) item

LR(0) item is the elementary building block of LR(0) parser, which we will be using extensively throughout this whole chapter.

Definition 4.1.1. *LR(0) item* is a pair of production rule and a position in the right-hand side of the production rule, where $0 \leq position \leq |\text{right-hand side}|$ must hold.

LR(0) items in this text will be denoted as the production rule with a $\cdot$ representing the position in the right-hand side of the production rule.

Example 4.1.1. LR(0) item $(A \to \alpha B \beta, 1)$ will be denoted as $A \to \alpha \cdot B\beta$

Informally, the position states how much of the right-hand side of the production rule we have already seen. Everything to the left of the $\cdot$ is on the stack. Symbol to the right is expected to be read from the input next. If the $\cdot$ is at the end of the production rule, it means we have the whole right-hand side of that production rule on the stack, and we can reduce it. I will be omitting the classifier LR(0) when it is obvious we are talking about LR(0) items.

We represent the state of the LR(0) automaton by the set of these LR(0) items.

Name: getLR0Closure
Input: Context-free grammar G, set of LR(0) items I
Output: set of LR(0) items I'

I' $\leftarrow$ I

do
  for each item A $\rightarrow \alpha \cdot$Bβ in I' (B **is** nonterminal)
    for each production rule B $\rightarrow \gamma$ of G
      I' $\leftarrow$ I' $\cup$ {B $\rightarrow \cdot\gamma$}
while new items were added to I' in this iteration

Algorithm 4.1: LR(0) closure [3, p. 245]

4.2 LR(0) closure

When the $\cdot$ is in the in front of a nonterminal in the LR(0) item, it means we expect to read next whichever symbol the nonterminal can be derived to. To complete the state, we will add all of the possible rules that can be used to derive that nonterminal. We will call this operation *LR(0) closure*, and it is formally defined by Algorithm 4.1. When talking about LR(0) closure on the LR(0) items, I might omit the classifier *LR(0)*.

Example 4.2.1. Let's say we have a set of LR(0) items $I = \{E \rightarrow \cdot T\}$ from the expression grammar. The LR(0) closure of I would look like this:

$$\{E \rightarrow \cdot T,\ T \rightarrow \cdot T * F,\ T \rightarrow \cdot F,\ F \rightarrow \cdot(E),\ F \rightarrow \cdot id\}$$

Definition 4.2.1. *Kernel items* are all LR(0) items whose position is not zero. One exception is the initial item of the augmented grammar $S' \rightarrow \cdot S$, which is also a kernel item.

4.3 Augmented grammar

Definition 4.3.1. Let G be a context-free grammar, $G = (N, \Sigma, P, S)$. Grammar $G' = (N \cup \{S'\}, \Sigma, P \cup \{S' \rightarrow S\}, S')$, where $S' \notin N \cup \Sigma$, is called the *augmented grammar* of G.

The reason for this augmentation is to tell the parser when to accept the input. We should only accept the input when we are ready to reduce by the new initial production rule.

Example 4.3.1. The augmented grammar EG' of the expression grammar would be:

$$
\begin{aligned}
&EG' = (\{E', E, T, F\}, \{+, *, (,), id\}, P', E') \\
&P' = \{ \\
&\quad E' \to E, \\
&\quad E \to E + T \mid T, \\
&\quad T \to T * F \mid F, \\
&\quad F \to (E) \mid id, \\
&\}
\end{aligned}
$$

4.4 LR(0) automaton

States of the LR(0) automaton are uniquely defined by their set of LR(0) items.

4.4.1 Initial state

Initial state consists simply of the closure of the new initial item from the augmented grammar $S' \to \cdot S$. What this means is that we haven't read any input yet, and there are no other states on the stack, but the initial state.

4.4.2 Transitions and next states

To figure out the transitions to the next state, we will look at every item, and whenever the position is smaller than the size of right-hand side (the $\cdot$ is not in the rightmost position of the right-hand side), we will look to the next symbol, and add transition over that symbol. The next state will contain that LR(0) item with position increased by one. If there is multiple items with the same symbol on the next position, we will merge them all into one state. These will be the kernel items of the next state, and we will call the closure on them, for the state to be complete. The algorithm for generating next state for certain LR(0) items and symbol is shown in Algorithm 4.2.

Example 4.4.1. LR(0) automaton for the expression grammar is shown in Figure 4.1. I_0 is the start state. Notice that I_0 contains only one kernel item, $E' \to \cdot E$, and the rest is generated by the closure function.

Algorithm for creating the LR(0) automaton is formally defined in Algorithm 4.3, and it was inspired by [3, p. 246], but they work with a canonical collection of sets of LR(0) items, instead of an automaton.

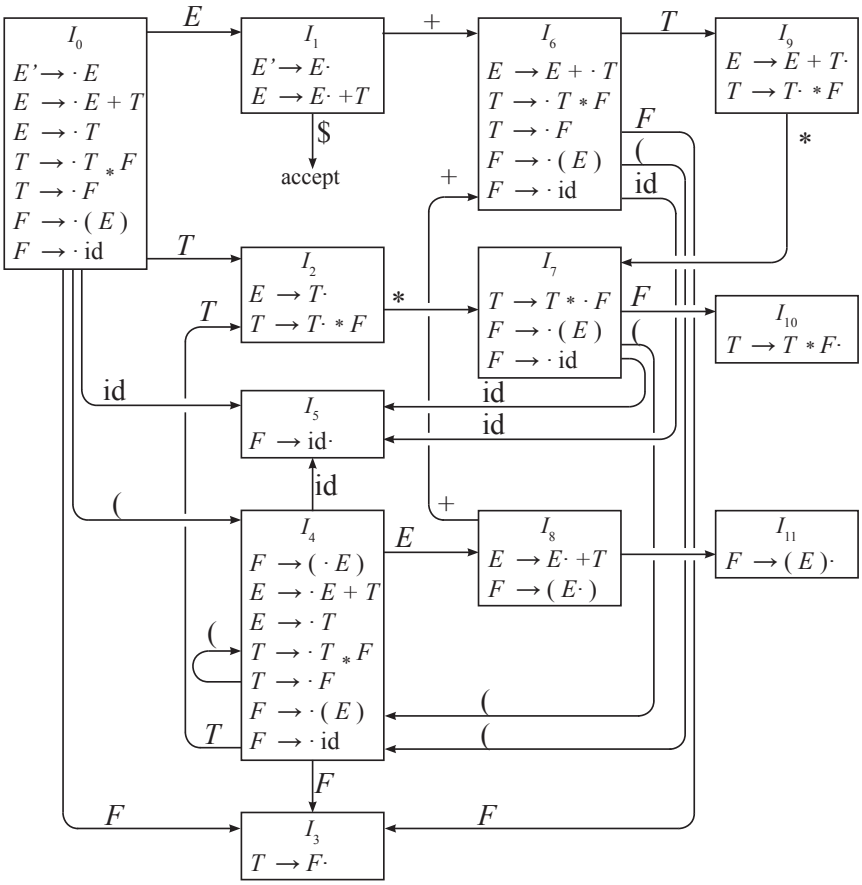


Figure 4.1: LR(0) automaton for the expression grammar [3, p. 244].

```
Name: getLR0NextStateItems
Input: LR(0) items currentItems
       symbol over
       context-free grammar G
Output: LR(0) items nextItems

let G' ← (N', Σ', P', S') be the augmented grammar of G

for every item p in currentItems
  if p is in form A → α ·, α ∈ (N' ∪ Σ')*
    continue

  let p be in form A → α · over β, α ∈ (N' ∪ Σ')*, β ∈ (N' ∪ Σ')*
  add A → α over · β to nextItems

nextItems ← getLR0Closure(nextItems)
```

Algorithm 4.2: LR(0) next state generator

4.5 SLR(1) parsing

SLR(1) stands for **S**imple **LR(1)**, and since we have one symbol of lookahead, it is more powerful than LR(0). This means that while creating the parsing tables, we can look at one symbol ahead to make decision whether to shift or reduce, and by what production rule to reduce by. SLR(1) parser can parse all grammars the LR(0) can. It is important to notice that the lookahead is used while creating the parsing tables, not while parsing the input string.

Compared to other LR algorithms, SLR(1) parsers can parse all grammar the LR(0) parser can, and "SLR(1) parsers are intermediate in power between LR(0) and LALR(1). Since SLR(1) parsers have the same size as LALR(1) parsers but are considerably less powerful, LALR(1) parsers are generally preferred." [6].

Informally, when the SLR(1) parser encounters the possibility to reduce by production rule $A \to \alpha$ (α is a string), it looks at the next input symbol. If the symbol could follow after A (using the Follow sets [4] from LL(1) parser), the parser does the reduction.

SLR(1) uses the LR(0) automaton for efficient creation of the action and goto tables.

4.5.1 Action table

Algorithm for creating the SLR(1) action table is described in Algorithm 4.4.

```
Name: getLR0Automaton
Input: Context-free grammar G
Output: LR(0) automaton A ← (Q, Σ, δ, q0, F)

let G' ← (N', Σ', P', S') be the augmented grammar of G
let S be start symbol of G

initialState ← getLR0Closure({S' → · S}, G')

Q ← {initialState}
Σ ← N' ∪ Σ' ∪ {$}
q0 ← initialState

queue.push(initialState)
while queue is not empty
  C ← queue.pop()

  for every symbol s ∈ N' ∪ Σ'
    nextState ← getLR0NextStateItems(C, s, G)

    if (nextState is not empty)
      if (nextState is not in A)
        Q ← Q ∪ {nextState}
        queue.push(nextState)
      δ(C, s) ← nextState
```

Algorithm 4.3: LR(0) automaton construction

4.5.2 Goto table

Algorithm for creating SLR(1) goto table is fairly easy. For every transition from state I_c to I_n over nonterminal A in the corresponding LR(0) automaton, we set $Goto[I_c, A] = I_n$. This algorithm is formally described in Algorithm 4.5.

```
Name: getSLR1ActionTable
Input: Context-free grammar G
Output: SLR(1) action table

let G' ← (N', Σ', P', S') be the augmented grammar of G
let A ← (Q, Σ, δ, q0, F) be the LR(0) automaton of G

for every state q in Q
  for every LR(0) item I in q
    if I is in format A → α·
      if A is S'
        Action[I, A] ← Accept
      else
        for every symbol f in Follow(A)
          Action[I, f] ← Reduce[A → α·]
    else if I is in format A → α·aβ  (a ∈ Σ')
      if δ(I, a) is not empty
        Action[I, a] ← Shift[δ(I, a)]
```

Algorithm 4.4: SLR(1) action table construction

```
Name: getSLR1GotoTable
Input: Context-free grammar G
Output: SLR(1) goto table

let G' ← (N', Σ', P', S') be the augmented grammar of G
let A ← (Q, Σ, δ, q0, F) be the LR(0) automaton of G

for every state q in Q
  for every nonterminal A in N'
    if δ(I, A) is not empty
      Goto[I, A] ← δ(I, A)
```

Algorithm 4.5: SLR(1) goto table construction

Chapter 5

Analysis

5.1 Existing solutions

There are no direct competitors, since LR(0) and SLR(1) are both fairly weak parsing techniques, but there is a lot of solutions using other LR, or even some custom parsing algorithms.

5.1.1 ANTLR

ANTLR [7] stands for **AN**other **T**ool for **L**anguage **R**ecognition and it is "a powerful parser generator for reading, processing, executing, or translating structured text or binary files. It is widely used to build languages, tools, and frameworks. From a grammar, ANTLR generates a parser that can build and walk parse trees." [7]

ANTLR uses custom ALL(*) [8] parsing algorithm, which is an extension of LL(*) algorithm, and it analyzes the grammar on-the-fly, instead of doing a static analysis. ANTLR can parse all grammars with the exception of grammars containing indirect left recursion.

5.1.2 Yacc

Yacc [9] stands for **Y**et **A**nother **C**ompiler-**C**ompiler and it accepts LALR(1) grammars with disambiguating rules. Yacc was very popular, and often distributed by default with Unix systems.

5.1.3 Bison

GNU Bison [10] (Bison for short) is a parser generator (compiler-compiler) which was written as a replacement for Yacc. It works with context-free grammars, and generates either LALR(1) or GLR parser. Support for IELR(1) [11] and LR(1) parser is in experimental stage. Bison is compatible with Yacc in a way that it accepts grammar definitions in the Yacc format.

5.1.4 LRSTAR

LRSTAR is a C++ parser generator. In the past, LRSTAR supported multiple parsing algorithms (LALR(1), canonical LR(1), minimal LR(1) and nondeterministic LR(k)). Since version 7.0, only minimal LR(k) is supported, as it has the best properties (fast and small) and as Paul Mann (author of LRSTAR) says, "too many choices can get confusing" [12].

5.1.5 Hyacc

Hyacc [13] (stands for Hawaii Yacc) is a parser generator which supports LR(1), LALR(1) and LR(0) parsing methods. It is interesting because of the support of the LR(0) parsing method, which is a part of this work, and is not usually supported by parser generators.

5.2 Automata library

The *Automata library* is a collection of multiple binaries and dynamic libraries. It is written in C++11, and the binaries are following the Unix ideology—each one does a single operation, and they can be chained using Unix pipes. They communicate with each other using a custom XML protocol [14], but lately we have been thinking about replacing it with a binary one, since the XML one is not the best fit. The Automata library contains mostly data structures and algorithms for working with strings, grammars, regular expressions, automata, and graphs.

The Automata library was created as a part of Martin Žák's bachelor's thesis [14], and it was later extended by Jan Veselý [15], Tomáš Pecka [16], Štěpán Plachý [17] and David Rosca [18] in their respective bachelor's theses. The Automata library is under active development, lead by Ing. Jan Trávníček.

There are eight dynamic libraries: *libalib2algo.so*, *libalib2common.so*, *libalib2data.so*, *libalib2elgo.so*, *libalib2measurepp.so*, *libalib2raw.so*, *libalib2std.so*, and *libalib2str.so*.

In past, the Automata library contained only implementation of LL(1) parsing [4], which is not very strong, therefore we decided to implement some more powerful LR parsing algorithms. My work will neither change nor add any binaries. I will add types to *libalib2data.so* and implement algorithms in *libalib2algo.so*.

I used multiple data structures and algorithms already implemented in the Automata library. I used classes for representing symbols (*alphabet::Symbol*), context-free grammars (*grammar::CFG*), and deterministic finite automata (*automaton::DFA*). I also used overloads for generating XML representation of LR(0) items, since the set of LR(0) items is implemented using only the standard containers and primitive types. I used the *label::LabelBase* class as a base class for my own label, used to represent the set of LR(0) items in

the *automaton::State*. The last thing I used was the algorithm for generating Follow sets [4].

Chapter 6

Implementation

The implementation is divided into three type definitions, one enumeration, and four classes. The type definitions and the enumeration are part of the *libalib2data.so* dynamic library, and the classes are a part of the *libalib2algo.so* dynamic library. All of these structures reside in the *grammar::parsing* namespace.

6.1 Enumeration LRAction

Enumeration *LRAction* is used in LRActionTable to represent the three shift-reduce actions—*Shift*, *Reduce*, and *Accept*.

6.2 Type definitions

It is usual in the Automata library not to have data encapsulated in classes, but rather work directly with raw containers and primitives. Since nested data type definitions can get very long, it is better to make a type definition using the C++ *typedef* declaration. There are three type definitions I added to the Automata library—*LR0Items*, *LRActionTable*, and *LRGotoTable*.

LR0Items represents a set of LR(0) items. It was inspired by how set of production rules is represented inside the Automata library, I just added the position needed for LR(0) items. LRActionTable, as the name suggests, represents an action table used in LR parsing. The underlying data structure is a map, which has a pair of state and symbol as a key (representing the row and column key respectively), and pair of LRAction and variant as a value. Contents of the variant depend on the value of LRAction:

- If the LRAction is Shift, it contains the state to be pushed onto the stack.

- If the LRAction is Reduce, it contains the production rule to reduce by. We pop |right-hand side| items off the stack. What will be pushed onto the stack is decided by the left-hand side, current top of the stack, and the goto table.

- If the LRAction is Accept, contents of the variant are undefined, since we don't need to use them.

In Algorithm 3.1, error state is considered to be an independent action, but to save memory, we represent the error simply by the absence of the entry in the table.

LRGotoTable represents a goto table used in LR parsing, and it has similar structure as the LRActionTable. The underlying data structure is also a map and the key is also the same. The only part where LRGotoTable and LRActionTable differ is the value, where LRGotoTable has only a single state.

The type definitions for action and goto tables follow the earlier pseudocodes as closely as possible, but if needed, they could be easily optimized for memory usage. We don't actually need to save the states as a whole, but we could assign them numbers, and identify them by those numbers, which would save a lot of memory. The LR0Items could be optimized similarly—we could assign numbers to the rules in the grammar, and save just the position and number identifier for each item. Although for this to be time efficient, it might be needed to change how rules are stored in *grammar::CFG*.

6.3 Class LRParser

Class LRParser contains three functions which are universal to all LR parsing methods.

6.3.1 Function *getEndOfInputSymbol*

This function is used to generate unique end-of-input symbol for given context-free grammar. It starts with symbol $, which is used in examples throughout this work, and if the symbol is already in the terminal/nonterminal alphabet of the grammar, it transforms it until it is unique.

6.3.2 Function *getAugmentedGrammar*

This is an implementation of algorithm described in Definition 4.3.1. To ensure the uniqueness of the new start symbol in the original grammar, we use the *alphabet::Symbol::next()* method to transform the start symbol until it is unique.

6.3.3 Function *parse*

This is universal parsing function for all LR methods as described in Algorithm 3.1. One change to the pseudocode is that we don't treat error state as an independent LRAction, but we check for the absence of value in the table. The function returns *true* if the string is parsable by the provided action and goto tables, and *false* otherwise.

In the future, it might be desirable to change the return value to a list of shift-reduce actions, or to pass a callback which will be called on every LR action. The current implementation is general enough to be suitable for these changes.

6.4 Class LR0Parser

6.4.1 Function *getClosure*

Function *getClosure* is an implementation of the Algorithm 4.1.

6.4.2 Function *getNextStateItems*

This function is an implementation of the Algorithm 4.2.

6.4.3 Function *getAutomaton*

This is an implementation of Algorithm 4.3. It could be optimized for memory usage, by saving only the kernel items. The rest can be generated lazily using the LR(0) closure function.

6.5 Class SLR1ParseTable

6.5.1 Function *getActionTable*

This function is an implementation of the Algorithm 4.4. Exception is thrown when provided grammar is not parsable by SLR(1) grammar. It uses LR(0) automaton to efficiently build the action table. This function also needs to generate a LL(1) Follow set [4], which is already implemented in the Automata library.

6.5.2 Function *getGotoTable*

This function is an implementation of the Algorithm 4.5.

6.6 Class LR0ItemsLabel

This class is a wrapper for LR0Items, used to label the states of the LR(0) automaton.

6.7 Testing

Testing was done using a series of unit tests. I tested the correctness of LR(0) automaton and correctness of action and goto tables. Using the generated parsing tables, I tested the universal LR parsing algorithm against series of both valid and invalid strings.

Conclusion

I thoroughly studied LR parsing in general, and LR(0) and SLR(1) parsing algorithms. I also did some research on more advanced algorithms like LL(*), LALR(1), LR(1), and GLR, so I could compare them, see the advantages and disadvantages, and understand their usage in other implementations of parser generators.

I implemented utility functions for augmenting a context-free grammar to our needs, generating closure of LR(0) items, function for generating the set of LR(0) items for the next state, and the algorithm for generating LR(0) automaton from a context-free grammar. The LR(0) automaton is used to generate SLR(1) action and goto parsing tables. Last algorithm I implemented is the universal LR parser, which accepts action table, goto table, and input string, and states if the string is parsable using those tables. The code is easily extensible, so if we decide later that we want to return the whole sequence of shift-reduce actions, it won't be a problem.

Last part of my work was testing, which was done using a series of unit tests covering all the implemented algorithms.

These algorithms can be used in the future as a foundation for more powerful algorithms like LALR(1) and LR(1).

Bibliography

[1] Rubiyath, A. DFA / NFA to Regular Expression (without using GNFA). 2011, [accessed 2016-04-02]. Available from: http://www.itsalif.info/content/dfa-nfa-regular-expression-without-using-gnfa

[2] McPeak, S. Elkhound: A Fast, Practical GLR Parser Generator. 2002.

[3] Aho, A. V.; Lam, M. S.; Sethi, R.; et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006, ISBN 0-321-48681-1.

[4] Begel, A. LL(1) Parsing. [accessed 2016-05-09]. Available from: http://research.microsoft.com/en-us/um/people/abegel/cs164/ll1.html

[5] Dill, D. L. CS143 Notes: Parsing. 1998—2015, [accessed 2016-05-10]. Available from: https://web.stanford.edu/class/archive/cs/cs143/cs143.1156/handouts/parsing.pdf

[6] Grune, D.; Jacobs, C. J. *Parsing Techniques: A Practical Guide*. Springer, 2008, ISBN 978-0-387-20248-8.

[7] Parr, T. ANTLR. 2014, [accessed 2016-05-08]. Available from: http://www.antlr.org

[8] Parr, T. *The definitive ANTLR 4 reference*. Dallas, Texas: The Pragmatic Bookshelf, 2013, ISBN 978-1-93435-699-9.

[9] Johnson, S. C. Yacc: Yet Another Compiler-Compiler. [accessed 2016-05-08]. Available from: http://dinosaur.compilertools.net/yacc/index.html

[10] Bison - GNU Project - Free Software Foundation. 2014, [accessed 2016-05-08]. Available from: https://www.gnu.org/software/bison/

[11] Denny, J. E.; Malloy, B. A. IELR(1): Practical LR(1) Parser Tables for Non-LR(1) Grammars with Conflict Resolution. 2008.

[12] LRSTAR News and Release Information. 2014, [accessed 2016-05-08]. Available from: `https://web.archive.org/web/20160326093421/http://lrstar.org/news.html`

[13] Chen, X. Hyacc. 2014, [accessed 2016-05-09]. Available from: `http://hyacc.sourceforge.net`

[14] Žák, M. *Automatová knihovna – vnitřní a komunikační formát.* Bachelor's thesis, Czech Technical University in Prague, 2014.

[15] Veselý, J. *Automatová knihovna – determinizace konečných a zásobníkových automatů.* Bachelor's thesis, Czech Technical University in Prague, 2014.

[16] Pecka, T. *Automatová knihovna – převody mezi regulárními výrazy, regulárními gramatikami a konečnými automaty.* Bachelor's thesis, Czech Technical University in Prague, 2014.

[17] Štěpán Plachý. *Automatová knihovna - Stromové automaty a algoritmy nad stromy.* Bachelor's thesis, Czech Technical University in Prague, 2015.

[18] Rosca, D. *Automatová knihovna - isomorfismus planárních grafů.* Bachelor's thesis, Czech Technical University in Prague, 2016.

Appendix A

List of used abbreviations

ANTLR	Another Tool for Language Recognition
CFG	Context-Free Grammar
DFA	Deterministic Finite Automaton
GLR	Generalized LR
GNU	GNU is a recursive acronym for "GNU's Not Unix!"
IELR	Inadequacy Elimination LR
LL	First L states that we read the input from left to right and the second L states that we perform a leftmost derivation
LR	L states that we read the input from left to right and R states that we construct a rightmost derivation in reverse
LALR	Look-Ahead LR
SLR	Simple LR
XML	Extensible Markup Language
Yacc	Yet Another Compiler-Compiler

Appendix B

Contents of enclosed memory card

- `readme.txt` ..... description of the contents of this memory card
- `source`
  - `automata-library` ..... source code of the Automata library
  - `thesis` ..... LaTeX source code of this thesis
- `text`
  - `thesis.pdf` ..... this thesis in the PDF format